Common Lisp Music update report
Fernando Lopez-Lezcano, Juan Pampin
Center for Computer Research in Music and Acoustics (CCRMA), Stanford
University

Abstract

In this paper we will present an update on the CLM sound synthesis and processing package.

1 Introduction

Common Lisp Music is a music synthesis and signal processing package developed by William
Schottstaedt at CCRMA over the last 10 years. This report is based on a snapshot of the current
state (as of this writing) of CLM-2, a new version of the package. CLM-2 represents a large-
scale revision of the previous version [1][2].

CLM exists now in three forms: a Common Lisp implementation with an optional C foreign
function interface, a pure C version, and a Scheme version built on the C version through
the Guile library (Guile is an implementation of the Scheme programming language). There
are a variety of unavoidable differences between these versions, but in general, the differences
are obvious and consistent. The Common Lisp and Scheme versions try to be completely com-
patible, and this paper focuses on the former.

CLM supports multichannel soundfile play-
back and creation and reads and writes a
large number of soundfile formats. CLM
currently runs on the NeXT (68k or Intel),
Macintosh (MacOS or MkLinux/LinuxPC),
SGI, Sun, Alpha, Windows, and any ma-
chine running Linux. It is available free, via anonymous ftp at "ftp://ccrma-
ftp.stanford.edu/pub/Lisp/clm.tar.gz". CLM
has been used for more than ten years by
composers, sound designers, and researchers.

2 Instruments

The CLM instrument design language is a subset of Common Lisp, extended with a large lib-
rary of sound-related functions and generators. In the Common Lisp implementation, CLM in-
struments can run in pure lisp, or can be auto-
matically translated and compiled into opti-
mized C code.

2.1 definstrument

The normal structure of a CLM instrument as
defined in Common Lisp is:

(definstrument name (arguments)
  (setup-code
    (run run-time-code)))

The setup code creates any needed generator
structures for the run-time code, which actu-
ally generates the samples. The run-time code
can contain a subset of the standard Common
Lisp functions, CLM generators, or other spe-
cial functions described latter. The run macro
normally translates its body from Lisp to C to
get a substantial speed-up in computation time
(currently it is only possible to use one call of
run per instrument definition).

(definstrument simp (start-time duration
  frequency amplitude
  key (amp-env '(0 0 80 1 100 0)))
(multiple-value-bind (beg end)
  (times->samples start-time duration)
  (let ((a (make-oscil :frequency frequency))
    (amp (make-oscil :envelope amp-env
                     :scaler amplitude
                     :duration duration)))
    (run
      (loop for i from beg to end do
        (outa i (+ (amp env amp) (oscil s)))))

"simp" is a very simple instrument that plays
a sine wave tone with an amplitude envelope.
CLM comes with a wealth of example instruments that illustrate many synthesis and signal processing techniques (additive synthesis, fm, waveshaping, convolution, filtering, granular synthesis, reverberation, physical modeling, spectral modeling, etc).

2.2 defpinstrument

Defpinstrument is the same as definstrument except that in addition to producing C output the instrument is handled by a special scheduler that runs it one sample at a time, in parallel with all other p-instruments that want to run on that sample. This kind of scheduling is useful when instruments share global data that they are modifying as they run, or for real-time sound output (see with-psound below). P-instruments can be mixed freely with "normal" instruments. The latter run to completion when they are called, while the currently running p-instruments wait in the background.

3 Generators

Each CLM generator consists of a set of two functions. Make-<gen> sets up the data structure associated with the generator at initialization time, <gen> runs the generator producing a new sample each time it is called and <gen>? checks whether a variable is that kind of generator. For example:

(setf oscillator (make-oscil :frequency 330))

prepares oscillator to produce a sine wave when set in motion via

(oscil oscillator)

Table 1 lists all available CLM generators with a short description of their functions.

It goes without saying that new generators can be easily implemented through the use of Common Lisp macros [3]. If you want to optimize your code for speed it is also possible to create C language generators and link them to CLM.

Special functions of interest include an fft facility that includes fft, make-fft-window (which supports a wide variety of predefined windowing functions), multiply-arrays, rectangular->polar and spectrum.

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-pass</td>
<td>all pass filter</td>
</tr>
<tr>
<td>amplitude-modulate</td>
<td>amplitude modulation</td>
</tr>
<tr>
<td>asymmetric-fm</td>
<td>asymmetric fm</td>
</tr>
<tr>
<td>comb</td>
<td>comb filter</td>
</tr>
<tr>
<td>contrast-enhancement</td>
<td>modulate sound</td>
</tr>
<tr>
<td>convolve</td>
<td>convolution</td>
</tr>
<tr>
<td>delay</td>
<td>delay line</td>
</tr>
<tr>
<td>env</td>
<td>line segment envelope</td>
</tr>
<tr>
<td>filter</td>
<td>direct form FIR/IIR filter</td>
</tr>
<tr>
<td>FIR-filter</td>
<td>FIR filter</td>
</tr>
<tr>
<td>IIR-filter</td>
<td>IIR filter</td>
</tr>
<tr>
<td>formant</td>
<td>resonance</td>
</tr>
<tr>
<td>granulate</td>
<td>granular synthesis</td>
</tr>
<tr>
<td>in-anything</td>
<td>sound file input</td>
</tr>
<tr>
<td>locsig</td>
<td>static sound placement</td>
</tr>
<tr>
<td>notch</td>
<td>notch filter</td>
</tr>
<tr>
<td>one-pole</td>
<td>one pole filter</td>
</tr>
<tr>
<td>one-zero</td>
<td>one zero filter</td>
</tr>
<tr>
<td>oscil</td>
<td>sine wave and FM</td>
</tr>
<tr>
<td>out-anything</td>
<td>sound output</td>
</tr>
<tr>
<td>polynomial</td>
<td>polynomial</td>
</tr>
<tr>
<td>pulse-train</td>
<td>pulse train</td>
</tr>
<tr>
<td>rand, rand-interp</td>
<td>random numbers, noise</td>
</tr>
<tr>
<td>readin</td>
<td>sound input</td>
</tr>
<tr>
<td>ring-modulate</td>
<td>ring modulation</td>
</tr>
<tr>
<td>buffer</td>
<td>block oriented processing</td>
</tr>
<tr>
<td>sawtooth-wave</td>
<td>sawtooth oscillator</td>
</tr>
<tr>
<td>sine-summation</td>
<td>sine summation synthesis</td>
</tr>
<tr>
<td>square-wave</td>
<td>square wave oscillator</td>
</tr>
<tr>
<td>src</td>
<td>sample rate conversion</td>
</tr>
<tr>
<td>sum-of-cosines</td>
<td>band limited pulse train</td>
</tr>
<tr>
<td>table-lookup</td>
<td>interpolated table lookup</td>
</tr>
<tr>
<td>tap</td>
<td>delay line tap</td>
</tr>
<tr>
<td>triangle-wave</td>
<td>triangle wave oscillator</td>
</tr>
<tr>
<td>two-pole</td>
<td>two pole filter</td>
</tr>
<tr>
<td>two-zero</td>
<td>two zero filter</td>
</tr>
<tr>
<td>wave-train</td>
<td>wave train</td>
</tr>
<tr>
<td>waveshape</td>
<td>waveshaping synthesis</td>
</tr>
</tbody>
</table>

Table 1: clm generators
3.1 Generic functions

It is often necessary to dynamically change parameters of a generator at run-time (an example would be changing the coefficients of a filter using envelopes). CLM provides a set of generic functions that can be applied to any generator that supports the underlying concept in a transparent way (ie: the generic function does "the right thing" for each supported generator).

All generic functions are self-able, where that makes sense, and apply to any structure that happens to support the notion in question. So, for example (setf (mus-frequency osc1) 440.0) sets osc1’s osc-freq field to (hz->radians 440.0). Similarly (read-position src-gen) returns the current position in the input file of src-gen.

3.2 Frames and Mixers

There are two special data types in CLM: frames and mixers. A frame is an array that represents a multi-channel sample (that is, in a stereo file, at time 0.0, there are two samples, one for each channel). A mixer is an array of arrays that represents a set of input and output scalers, as if it were the current state of a mixing console’s volume controls. A frame (a multi-channel input) can be "mixed" into a new frame (a multi-channel output) by passing it through a "mixer" (a matrix, the operation being a matrix multiply). These are combined with the notion of a "sample" (one datum of sampled music), and input/output ports (files, audio ports, etc) to handle all the underlying data.

4 Score

CLM does not have a specialized score language. Since CLM instruments are Lisp functions, a CLM note list can consist of any Lisp expression that calls those functions. This paradigm allows for very flexible control at the event level, where the user can write Lisp functions to generate sound events.

4.1 with-sound

With-sound is a macro that performs all the various services needed to create and play a sound file; it also wraps an "unwind-protect" around its body to make sure that everything is cleaned up properly if you happen to interrupt computation, then returns the output soundfile name.

(with-sound (:\output "new.snd")
(simp 0 1 440 .1)
(loop for st from 0 by 0.12
  for fr from 124.1 by 22
  repeat 4 do
    (simp st 0.8 fr .11)))

This snippet of code creates a new soundfile named "new.snd" and renders one explicit note of the simp instrument and four additional ones that are created from within a loop macro call. The list that follows the with-sound call can be used to specify global parameters such as output soundfile, soundfile type, data type, number of output channels, sampling rate, reverberation instrument and options, overall amplitude scaling to avoid clipping, etc.

Clm-load is the same as with-sound, but its first argument is the name of a file containing clm instrument calls (i.e. the body of with-sound)

4.2 with-mix

With-mix is a macro, callable within with-sound or clm-load, which saves the computation in its body in a separate file, and can tell when that file’s data is up to date and need not be recomputed.

(with-sound ()
  (fm-violin 0 .1 440 .1)
  (with-mix () "sec1" .5
    (fm-violin 0 .1 550 .1)
    (fm-violin .1 .1 660 .1))
  (with-mix (:reverb jn-reverb) "sec2" 1.0
    (fm-violin 0 .1 880 .1 :reverb-amount .2)))

Each with-mix in the preceding example saves its body within the header of the resulting intermediate soundfile so that it can compare the source code that generated it with a subsequent call to the same with-mix. It only recalculates its body if the source code has actually changed. Otherwise it mixes in the intermediate soundfile instead. It is easy to use with-mix to wrap unchanging sections of a piece so as to avoid unnecessary calculations (it functions in a way that is similar to the unix make facility).

4.3 Network rendering

If you have several machines connected by a reasonably fast net, and you’re running ACL (Allegro Common Lisp) or MCL (Macintosh Common Lisp), you can use a variant of with-sound
that parcels out the individual note calls to all the machines in parallel. It takes one additional argument: helpers, which is a list of host names that you want to use to compute the body.

4.4 \textit{with-psound}

\textit{With-psound} runs its body in real-time, in the sense that the sound is sent directly to the DAC as it is being computed. Obviously this only works if the computer is fast enough to calculate the body faster than real-time. Ordinary instruments can be handled by \textit{with-psound} if there's only one note in the body; for more complex cases you have to use p-instruments (defined by the \textit{defpinstrument} macro).

4.4.1 Real-time controls

Real-time control and value readback is handled in CLM through shared memory. Controls are defined as offsets within a shared memory block and can be accessed from the lisp world or any external programs that references it. Real-time controls can be defined in CLM by using lisp code that describes the binding of Motif sliders, buttons or MIDI messages to positions in the shared memory. The following code creates the interface for the waveform and fft tester illustrated in figure 1:

\begin{verbatim}
(make-controller "tester" nil
 '((tester-on "tester on/off" :toggle t)
  '(0 "":separator #""
    (test-wave "waveform" :graph)
  0 #:separator #"
    (tester_fft "fft" :graph))

A similar snippet of code creates an interface for a very simple ring modulator p-instrument as shown in figure 2. Two sliders control the sine wave oscillator frequencies and another the overall amplitude. The ring modulation p-instrument is connected to the previously defined tester. When the p-instrument is started, the tester interface displays the output time domain samples and their fft in real-time.

5 Conclusions

CLM-2 represents a big change in the structure of the language and is a work in progress. Evolutionary flotsam has been deleted, the name space has been cleaned, generators have been optimized and control functions generalized. The new C and Guile implementations open the way to embed CLM in other environments. In fact the trigger for this change was Bill's desire to include CLM within the Snd [4] sound editing environment.

References


   "http://www-ccrma.stanford.edu/CCRMA/
     Software/clm/clm-manual/clm.html"

   Sound Movement Toolkit" Proceedings of the
   ICMC 1994.

   "http://www-ccrma.stanford.edu/CCRMA/
     Software/snd/snd-manual/snd.html"